



Stiftung University of Hildesheim
Marienburger Platz 22
31141 Hildesheim
Germany



Software Systems Engineering (SSE)
Institute for Computer Science
Faculty for Mathematics, Natural Science,
Economics, and Computer Science



EASy-Producer

Engineering Adaptive Systems

Manual for the Build Server

Hildesheim, September 11, 2013

Contents

1	Introduction	4
2	Local Build	5
2.1	Installation of ANT	5
2.2	Relevant Folders	6
2.3	Preparation of Eclipse for compilation and testing	7
2.4	Editing <code>global-build.properties</code>	8
2.5	Building the Projects	9
3	Configuration of the Build Server	11
3.1	Editing Build Scripts	11
3.1.1	Managing Plug-in Dependencies	11
3.1.2	Creation of new Plug-ins	14
3.2	Configuring Jenkins	17
4	Frequently Asked Questions	20
4.1	Errors while compiling	20
4.1.1	Compile failed	20
4.2	Errors while testing	22
4.2.1	Could not find plug-in	22
4.2.2	<code>java.lang.NoSuchMethodError: com.vladium.emma.rt.RT.S</code>	24
4.2.3	<code>java.lang.NoClassDefFoundError</code>	25

List of Figures

1	Configuration of the Path variable for ANT	5
2	Relevant folders for compiling EASy-Producer	7
3	Local settings of the <code>global-build.properties</code>	8
4	Architecture of EASy-Producer	10
5	Manifest of the IVML editor	11
6	Settings of the VarModel project inside the <code>global-build.properties</code> . .	12
7	Build script of the IVML Editor	13
8	Build script of the IVML project	13
9	Build script of the IVML Editor (compilation target)	15
10	Build script of the IVML project (compile, instrument, copy targets)	16
11	Jenkins Logo	17
12	Jenkins: Creation of a new Build Job (Step 1)	17
13	Jenkins: Creation of a new Build Job (Step 2)	18
14	Jenkins: Creation of a new Build Job (Step 3)	19
15	Opening the OSGi console	22
16	Installation of a plug-in	23
17	Failed attempt of starting a plug-in	23
18	Added exclusion filter to solve a <code>com.vladium.emma.rt.RT.S</code> error	24
19	Creation of a jar, including referenced libraries	25

1 Introduction

This manual explains how to build the EASy-Producer tool suite on a local machine and also how to configure the build server, which can be accessed at <http://jenkins.sse.uni-hildesheim.de/>.

EASy-Producer consists of several Eclipse¹ *plug-ins*. Related plug-ins are stored together in the same folder on the Subversion² server. We denote these folders as *projects*. It is possible to build a complete project as well as a single plug-in via a build script. The build server builds all configured projects, which means that all nested plug-ins are built and tested in one step.

The remainder of this manual is structured as follows: Section 2 describes how to download and to build the individual projects and plug-ins. This also includes the installation of necessary build tools. This is needed to test the correctness of the build scripts. In Section 3, we explain how to maintain the existing build scripts. This also includes the configuration of the build server, which executes the build scripts. Finally, Section 4 shows some problems and how to solve them.

¹<http://www.eclipse.org>

²http://en.wikipedia.org/wiki/Apache_Subversion

2 Local Build

This section explains how to build EASy-Producer and all plug-ins on your local hard drive using ANT³. Section 2.1 roughly explains how to install ANT. Section 2.2 on the next page shows the relevant folders which must be downloaded to run the build scripts. Sections 2.3 and 2.4 explain local configurations, which must be done before running the build. Finally, Section 2.5 on page 9 explains the commands for running the build scripts on a local PC.

2.1 Installation of ANT

Download ANT from <http://ant.apache.org/> and extract the archive into the programs folder, e.g. C:\Program Files\Ant. We suggest ANT version 1.8.4, because this version is also used by the build server. Set the Path environment variable to the <ANT folder>\bin directory. Figure 1 shows an example how to configure the Path variable in Windows.

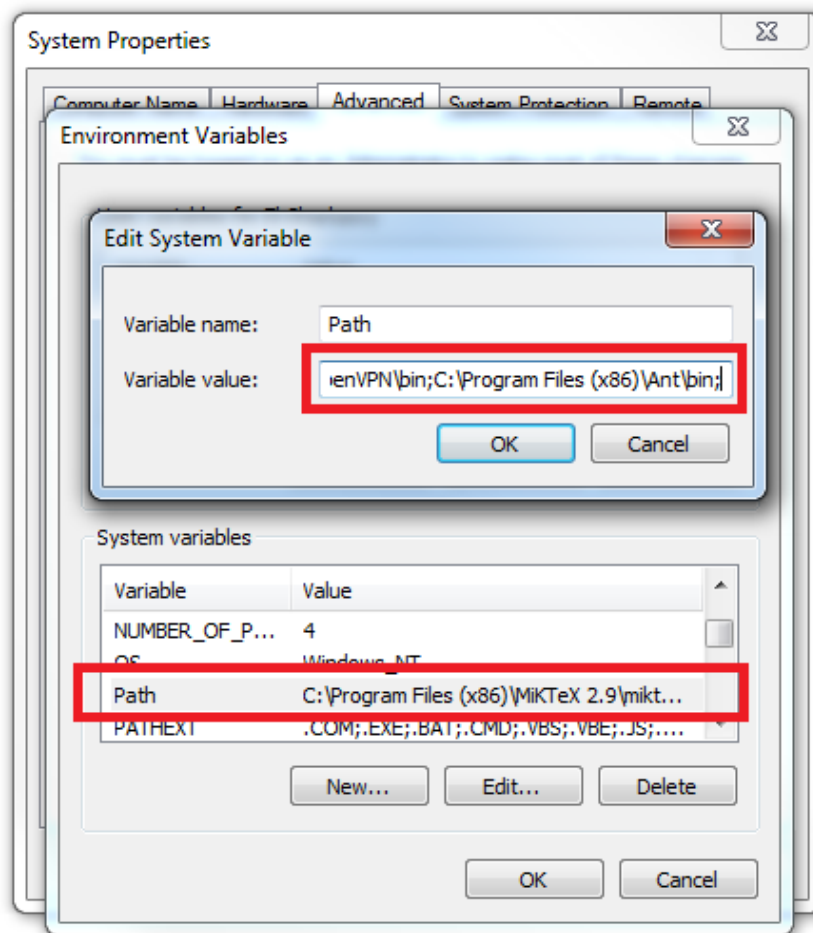


Figure 1: Configuration of the Path variable for ANT in Windows.

³<http://ant.apache.org/>

2.2 Relevant Folders

The whole EASy-Producer tool suite can be downloaded from <https://projects.sse.uni-hildesheim.de/svn/EASy/> via Subversion. The plug-ins are located in sub folders of `/trunk`. The documentation build script for the creation of a common JavaDoc is located in `/doc/javadoc`.

Relevant projects are:

/trunk/VarModel

The variability model and other utility functions which do not depend of other plug-ins.

/trunk/IVML

The parser and editor for the variability model.

/trunk/Instantiation

Tools and models needed for resolving variability in product line artifacts.

/trunk/Reasoner/ReasonerCore

Reasoner core functionality (interfaces and data objects). This package is not able to perform reasonings by its own and needs at least one of the reasoner implementations below:

/trunk/Reasoner/Drools

A reasoner implementation, using Drools Expert⁴.

/trunk/Reasoner/Jess

A reasoner implementation, using Jess⁵. Currently, not maintained.

/trunk/EASy-Producer

EASy-Producer core functionality and Eclipse editors.

It is also possible to check out the complete `/trunk` folder. The result should look like Figure 2 on the following page.

The `/trunk` folder contains also a `global-build.properties` file. This file must be copied to the HOME directory, e.g., the *user files* in Windows and edited as described in Section 2.4 on page 8.

⁴<http://www.jboss.org/drools/drools-expert.html>

⁵<http://www.jessrules.com/>

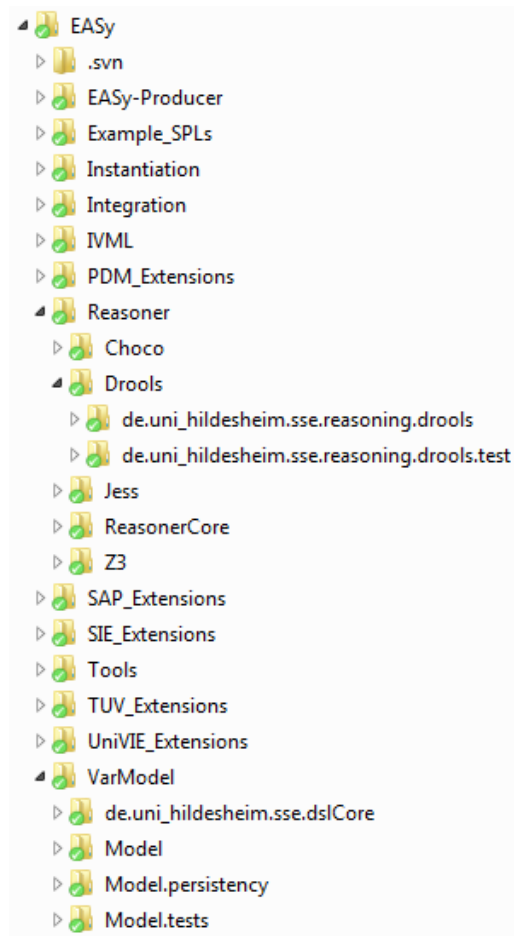


Figure 2: Relevant folders, which must be checked out, for compiling EASy-Producer.

2.3 Preparation of Eclipse for compilation and testing

For compilation and testing an Eclipse⁶ instance is needed, which contains needed plug-ins like Xtext⁷. We provide an already packed Eclipse instance, which can be used for this purpose, at <https://projects.sse.uni-hildesheim.de/eclipse/releases/EclipseTest.zip>. We suggest to use 2 instances:

- One unpacked instance for compilation. This Eclipse installation should not contain already compiled EASy-Producer plug-ins.
- One packed instance for testing. This Eclipse instance will be unpacked, before the test plug-ins will be installed into this instance. This procedure ensures a clean Eclipse installation for testing, which will also contain only needed EASy-Producer plug-ins.

⁶<http://www.eclipse.org/>

⁷<http://www.eclipse.org/Xtext/>

2.4 Editing `global-build.properties`

The copied `global-build.properties` (cf. Section 2.2 on page 6) must be edited to facilitate local builds. Figure 3 shows the relevant entries, which must be configured.

```
1 # Local settings
2 eclipse.home=${user.home}/Eclipse/EclipseCompile
3 alternative.test.eclipse.dir=${user.home}/Eclipse/EclipseTest
4 home.base.dir=${user.home}/jobs
5 projects.model.dir=VarModel/workspace/VarModel
6 projects.ivml.dir=IVML/workspace/IVML
7 projects.integration.dir=Integration/workspace/Integration
8 projects.reasonerCore.dir=ReasonerCore/workspace/ReasonerCore
9 projects.jess.dir=Jess/workspace/Jess
10 projects.drools.dir=Drools/workspace/Drools
11 projects.instantiation.dir=Instantiation/workspace/Instantiation
12 projects.easy.dir=EASy-Producer/workspace/EASy-Producer
13 emma.path=/var/lib/jenkins/additionalLibs/emma
14 test.eclipse.zip=${user.home}/Eclipse/EclipseTest.zip
15 unzipNewEclipse=true
```

Figure 3: Local settings of the `global-build.properties` (excerpt).

The entries should be configured as explained below:

eclipse.home

This entry must point to the unpacked Eclipse instance (**absolute path**), which shall be used for compilation.

home.base.dir

This entry must point to the root directory (**absolute path**) of the downloaded plug-ins, i.e., the downloaded `/trunk` folder.

projects.<project>

These entries must point to the **relative paths** of the sub folders of the related `<project>`s inside the `/trunk` folder.

emma.path

This entry must point to the **absolute path** of the EMMA libraries⁸. The development of Emma has been discontinued. As a consequence, we recommend to use the latest release for building EASy-Producer plug-ins.

⁸<http://emma.sourceforge.net/>

unzipNewEclipse

Specification whether a fresh Eclipse instance should be unpacked for each test (**true**) or not (**false**).

alternative.test.eclipse.dir

Absolute path of an unpacked Eclipse instance, which should be used for testing. Only relevant if **unzipNewEclipse** was set to **false**.

test.eclipse.zip

Absolute path of a packed Eclipse instance, which should be used for testing. Only relevant if **unzipNewEclipse** was set to **true**.

2.5 Building the Projects

Each of the relevant folders, listed in 2.2 on page 6, contain a **build.xml** file for building all related plug-ins in one step. All nested plug-ins contain a **build-jk.xml** for building the single plug-in only.

The build can be started by opening a command shell in the folder and running one of the commands below:

- Inside the project dir (e.g. VarModel): **ant**
- Inside the nested plug-in dir (e.g. VarModel/Model): **ant -f build-jk.xml**

Currently, the relationship as shown in Figure 4 on the next page exists between the plug-ins (and projects). For this reason, the plug-ins and projects should be build in the opposite order of the «*use*» relationships. For instance, the **Model** must be build before **IVML** can be build.

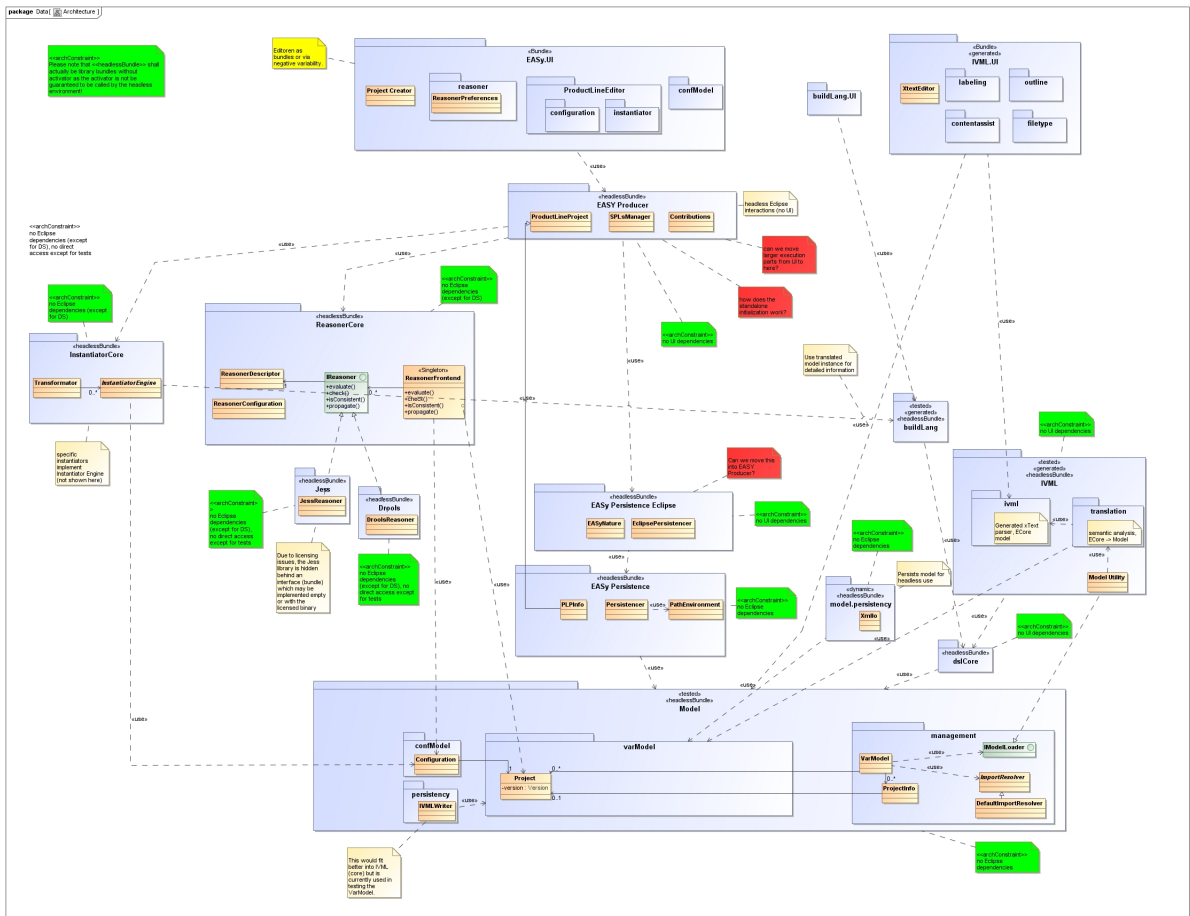


Figure 4: The architecture of EASy-Producer.

3 Configuration of the Build Server

This section explains how to configure the build server. First, we explain how to write and edit build scripts before we show how to configure Jenkins (<http://jenkins.sse.uni-hildesheim.de/>).

3.1 Editing Build Scripts

This section explains how build scripts should be written and edited, so that the build server can handle them. The `/trunk` folder is structured in projects containing multiple plug-ins, which are related to each other. For each project a sub folder inside of `/trunk` exist containing a `build.xml` for building the whole project, i.e., all related plug-ins. Section 3.1.1 describes how existing build scripts can be edited if dependencies between projects/plug-ins change. Section 3.1.2 shows how new plug-ins and projects can be created.

3.1.1 Managing Plug-in Dependencies

When the dependencies between plug-ins (OSGI Manifests) and/or projects change, multiple build files have to be changed manually. This section describes how to identify the plug-in dependencies and how to adapt existing build scripts.

```

1 Manifest-Version: 1.0
2 Bundle-ManifestVersion: 2
3 Bundle-Name: IVML User Interface (Editor)
4 Bundle-Vendor: University of Hildesheim - SSE
5 Bundle-Version: 0.4.1
6 Bundle-SymbolicName: de.uni_hildesheim.sse.ivml.ui; singleton:=true
7 Bundle-ActivationPolicy: lazy
8 Require-Bundle: de.uni_hildesheim.sse.ivml; visibility:=reexport,
9   org.eclipse.xtext.ui; visibility:=reexport,
10  org.eclipse.ui.editors,
11  org.eclipse.ui.ide,
12  org.eclipse.xtext.ui.shared,
13  org.eclipse.ui,
14  org.eclipse.xtext.builder,
15  org.antlr.runtime,
16  org.eclipse.xtext.common.types.ui,
17  org.eclipse.xtext.ui.codetemplates.ui,
18  org.eclipse.compare,
19  org.eclipse.core.resources,
20  org.eclipse.core.runtime,
21  de.uni_hildesheim.sse.varModel; bundle-version="0.0.3",
22  de.uni_hildesheim.sse.dslCore; bundle-version="0.0.1"
23 Import-Package: org.apache.commons.logging,
24   org.apache.log4j
25 aService-Component: OSGI-INF/contribution.xml
26 Bundle-RequiredExecutionEnvironment: J2SE-1.5
27 Export-Package: de.uni_hildesheim.sse.ui.contentassist,
28   de.uni_hildesheim.sse.ui.contentassist.antlr,

```

Figure 5: Manifest of the IVML editor plug-in (excerpt).

First it is necessary to identify the dependencies between the plug-ins of EASy-Producer. Plug-in dependencies to Eclipse, Xtext, ... will be resolved automatically. This can be done by reading the *manifest file* inside the plug-in. The manifest file is located at META-INF/MANIFEST.MF inside the plug-in directory. Figure 5 on the preceding page shows an example for a manifest file.

The sections **Require-Bundle** and **Import-Package** are important as they show plug-in dependencies.

Require-Bundle lists plug-ins, which are needed for the current plug-in.

Import-Package lists Java packages, which are needed for the current plug-in. Thus, it is unclear which plug-in is needed. Usually, plug-ins and packages follow the same naming conventions. Therefore, the package names give a hint, which plug-in is really needed.

The optional command **visibility:=reexport** allows plug-ins to provide functionality provided by imported plug-ins. Thus, also this command may hide needed plug-ins. Consequently, the manifest files of all imported plug-ins should also be checked to find all needed plug-ins.

An analysis of the manifest file in Figure 5 on the previous page indicates that the following plug-ins are needed among others:

- The variability model (de.uni_hildesheim.sse.varModel).
- The dsl core plug-in (de.uni_hildesheim.sse.dslCore)
- Some Eclipse plug-ins (org.eclipse.xtext.ui, org.apache.log4j, ...). These plug-ins need not be considered, as the build scripts will resolve them automatically.

Inside the **global-build.properties** two properties are defined for each plug-in:

- The property `home.<plug-in name>.dir` points to the directory of the plug-in.
- The property `libs.<plug-in name>` points to the jar file created by the build scripts.

Figure 6 shows an excerpt of the **global-build.properties**. This excerpt holds the definition of properties for the variability model.

```

1 # Settings related to project: Model
2 home.model.dir=${home.base.dir}/${projects.model.dir}/Model
3 libs.model=${home.model.dir}/${build.jar.dir}/de.uni_hildesheim.sse.varModel.
   jar
4 home.model.persistence.dir=${home.base.dir}/${projects.model.dir}/Model.
   persistence
5 home.model.tests.dir=${home.base.dir}/${projects.model.dir}/Model.tests
6 home.dslCore.dir=${home.base.dir}/${projects.model.dir}/de.uni_hildesheim.sse
   .dslCore
7 libs.dslCore=${home.dslCore.dir}/${build.jar.dir}/de.uni_hildesheim.sse.
   dslCore.jar

```

Figure 6: Settings of the VarModel project inside the **global-build.properties** (excerpt).

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project name="de.uni_hildesheim.sse.ivml.ui" default="jar" basedir=".">
3   <!-- import von globalen Properties Einstellungen -->
4   <property file="${user.home}/global-build.properties"/>
5
6   <!-- Einstellungen fuer dieses Projekt -->
7   <property name="src.dir" value="src"/>
8   <property name="src.gen.dir" value="src-gen"/>
9   <property name="javacSource" value="1.5"/>
10  <property name="javacTarget" value="1.5"/>
11
12  <path id="includes">
13    <!-- Model -->
14    <pathelement path="${libs.model}"/>
15    <!-- DSL core common classes -->
16    <pathelement path="${libs.dslCore}"/>
17    <!-- IVML Core -->
18    <pathelement path="${libs.ivml}"/>
19    <!-- Eclipse-Plugins -->
20    <fileset dir="${eclipse.plugins.dir}">
21      <include name="**/*.jar" />
22    </fileset>
23  </path>

```

Figure 7: Build script (build-jk.xml) of the IVML Editor (excerpt). `path="includes"` must be changed to import all needed plug-ins.

Whenever new plug-in dependencies occur (or old dependencies are removed) the build scripts have to be changed on different places. The `build-jk.xml` has to be changed for the plug-in itself and also the `build.xml` for the complete project containing the changed plug-in. We suggest first to update the `build-jk.xml` of the plug-in. Figure 7 shows the relevant parts, which must be changed. Inside the `includes` Section, new `pathelements` must be created whenever new plug-in dependencies occur.

```

1 <target name="copy.to.eclipse">
2   <copy todir="${test.eclipse.path}/plugins" failonerror="true" overwrite="
3     true">
4     <fileset dir="${coverage.instr.dir}/lib"/>
5     <fileset dir="${home.model.dir}/${build.jar.dir}" includes="**/*.jar"/>
6     <fileset dir="${home.reasonerCore.dir}/${build.jar.dir}" includes="
7       **/*.jar"/>
8     <fileset dir="${home.dslCore.dir}/${build.jar.dir}" includes="**/*.jar"
9     />
10  </copy>
11 </target>

```

Figure 8: Build script (build.xml) of the IVML project (excerpt). `target name="copy.to.eclipse"` must be changed to copy all needed plug-ins for testing the plug-ins of the project.

After the build scripts of the plug-ins were adapted, also the build script for the complete project must be changed. If the plug-in is tested than a target `copy.to.eclipse` exist, where all necessary plug-ins are copied into the *plug-ins folder* of the Eclipse instance for testing. Figure 8 on the preceding page shows an already prepared `copy.to.eclipse` target for the IVML project. In lines 3 – 6 necessary plug-ins are copied to eclipse. These lines must be changed to reflect the current plug-in dependencies. If the project is not tested, e.g. if it only consists of interfaces and so on, than the `build.xml` need not be changed.

3.1.2 Creation of new Plug-ins

This section describes necessary changes if new plug-ins (or projects) are created. If existing plug-ins should use the new plug-ins, than the adaptations described in the section before must be applied to the old plug-ins and projects. This section describes further changes necessary for the new plug-in. This consists of 4 steps explained in the remainder of this section:

1. Create a new project folder on the subversion server and upload all plug-ins of the project into the newly created folder.
2. Definition of new ant properties inside the `global-build.properties`.
3. The creation of a new `build-jk.xml` inside the plug-ins directory.
4. The creation/adaptation of a `build.xml` for the whole project.

After the new folder was created, new properties must be defined inside the `global-build.properties` file (cf. Figure 6 on page 12). Update the `globalbuild.properties` located in the `/trunk` folder. Create a new `home.<project name>.dir` property pointing to relative path of the plug-in's folder and create also a `libs.<project name>` property pointing to the jar file, which will be created after the execution of the plug-in's build script.

After the properties for the new plug-in were created, a new `build-jk.xml` must be created inside the plug-ins directory. The build files contain only as little individual code as possible. Therefore, the best way of creating a new `build-jk.xml` is coping an existing file from an old plug-in to the new plug-in's folder (we suggest to use the `build-jk.xml` from the Model plug-in) and modify the relevant passages:

1. Modify the name attribute of the project (cf. line 2 in Figure 7 on the preceding page). The name of the project is used for the creation of the jar file. Thus, the name of the project must match to the specified `libs.<plug-in name>` ant property.
2. Check whether the compilation settings are correct (cf. lines 6 – 10 in Figure 7 on the previous page). There, the right JDK version has to be selected (lines 9 and 10). Some projects contain also multiple source folders (lines 7 and 8). In this case, all relevant source folders should be defined in the section of compilation settings.
3. Edit the `includes` section to specify needed plug-ins for compilation (already described in Section 3.1.1 on the preceding page).
4. If multiple source folders exists and were specified in step 2, include the defined source folders inside the compilation target (cf. Figure 9 on the next page, line 4).

```
1 <!-- Compile all files without test classes -->
2 <target name="compile" depends="init">
3   <javac srcdir="${src.dir}" debug="on" destdir="${build.classes.dir}"
   includeAntRuntime="no" failonerror="true" source="${javacSource}" target=
   "${javacTarget}" encoding="${javac.encoding}">
4     <src path="${src.gen.dir}" />
5     <classpath refid="includes" />
6   </javac>
7 </target>
```

Figure 9: Build script (build-jk.xml) of the IVML Editor (excerpt). In line 4, the additional source folder `src.gen.dir` was included.

Finally, a `build.xml` must be created for the project. Also for this file is it the way to copy an existing file (we suggest to use the `build.xml` of the variability model) and to modify the relevant passages:

1. Inside the compilation target, all plug-ins of this project must be added, also the test plug-ins (cf. lines 2 – 15 in Figure 10 on the following page). Please use the `home.<plug-in name>.dir` for calling the `build-jk.xml` of the individual plug-ins.
2. Instrument plug-ins, which should be tested (cf. lines 22 – 24 in Figure 10 on the next page).
3. Copy all plug-ins needed for testing into the Eclipse instance for testing (cf. lines 32 – 35 in Figure 10 on the following page). In this step, all instrumented plug-ins as well as all plug-ins, which are needed for running the tested plug-ins, must be copied into the Eclipse instance.
4. Edit the target `coreTestEMMA` to call the test suite class inside the test plug-in.
5. Edit the target `emmaReport` to produce code coverage only for relevant source code files.
6. Finally, edit the target `javadoc` to create the Java documentation for relevant source code files.

```

1  <target name="compile">
2    <echo>#####</echo>
3    <echo>### Compiling IVML Parser ###</echo>
4    <echo>#####</echo>
5    <ant dir="${home.ivml.dir}" antfile="${build.script.name}"/>
6
7    <echo>#####</echo>
8    <echo>### Compiling IVML UI ###</echo>
9    <echo>#####</echo>
10   <ant dir="${home.ivml.ui.dir}" antfile="${build.script.name}"/>
11
12   <echo>#####</echo>
13   <echo>### Compiling IVML Tests ###</echo>
14   <echo>#####</echo>
15   <ant dir="${home.ivml.tests.dir}" antfile="${build.script.name}"/>
16 </target>
17
18 <target name="instrument">
19   <emma enabled="true" verbosity="verbose">
20     <instr destdir="${coverage.instr.dir}" metadatafile="${coverage.results
21       .dir}/metadata.emma" mode="fullcopy">
22       <instrpath>
23         <fileset dir="${home.ivml.dir}/${build.jar.dir}/" includes="**/*.
24           jar"/>
25         <fileset dir="${home.ivml.tests.dir}/${build.jar.dir}/" includes="
26           **/*.jar"/>
27         <!-- UI is not tested and thus not instrumented -->
28       </instrpath>
29     </instr>
30   </emma>
31 </target>
32
33 <target name="copy.to.eclipse">
34   <copy todir="${test.eclipse.path}/plugins" failonerror="true" overwrite="
35     true">
36     <fileset dir="${coverage.instr.dir}/lib"/>
37     <fileset dir="${home.model.dir}/${build.jar.dir}" includes="**/*.jar"/>
38     <fileset dir="${home.reasonerCore.dir}/${build.jar.dir}" includes="
39       **/*.jar"/>
40     <fileset dir="${home.dslCore.dir}/${build.jar.dir}" includes="**/*.jar"
41     />
42   </copy>
43 </target>

```

Figure 10: Build script (build.xml) of the IVML project (excerpt).

In this section, we explained how to create and modify the ant build scripts. With these scripts it is possible to build the newly created plug-ins on a local machine as well as on the build server. However, the build server is not able to detect newly created build scripts by its own. Therefore, also the build server has to be configured after new projects has been created. This is explained in the next section.

3.2 Configuring Jenkins



Figure 11: Jenkins Logo.

Currently, we use Jenkins (<http://jenkins-ci.org/>) as server for continuous integration. This server monitors the software configuration management system (SCM), i.e. Subversion, and triggers a new build whenever it detects changes in a project. Such a build includes the compilation of nested projects, testing including code coverage, the creation of JavaDoc, A build of a project may also lead to a build of related projects. This section describes how to configure Jenkins to build newly created projects (cf. Section 3.1).

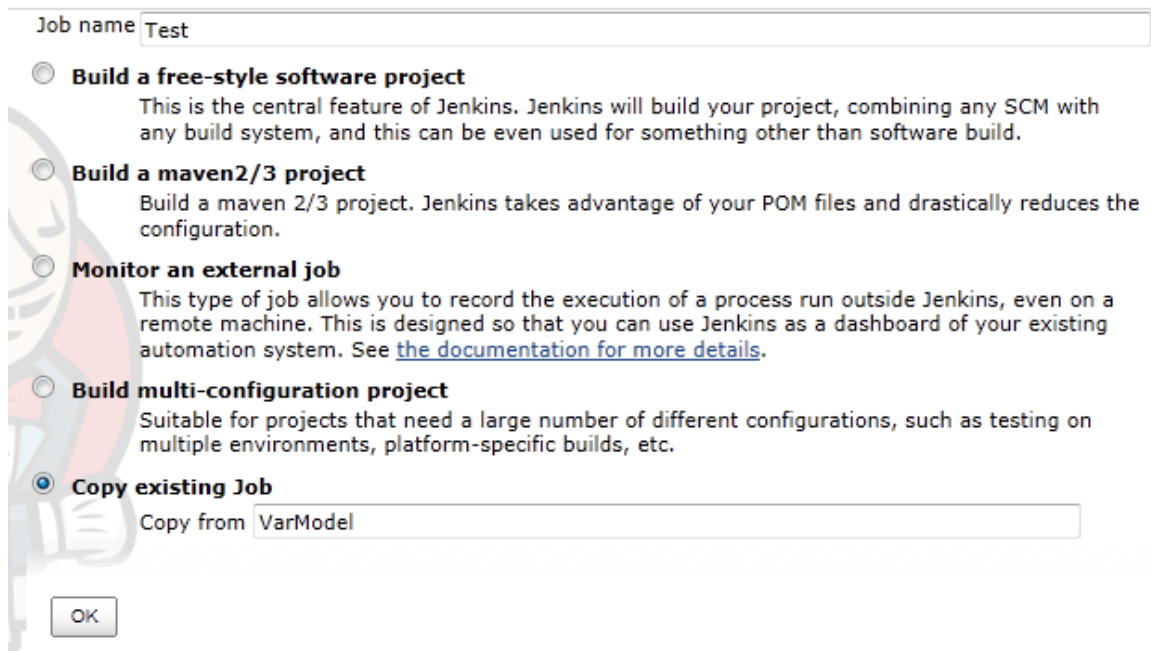
Jenkins is structured in **jobs**. The projects described above can be mapped directly to such jobs. Therefore, for each created project, a new job must be created in Jenkins. For doing so, a Jenkins account is needed. Please contact Sascha El-Sharkawy, if you have none.

After you are logged in, select "Jenkins" → "New Job" in the menu for creating a new build job (see Figure 12).



Figure 12: Creation of a new build job in Jenkins (step 1).

The next screen asks for a name and for the nature of the job. The easiest way of defining a new job is the selection of "Copy existing Job". An example is given in Figure 13 on the following page.



Job name

☐ **Build a free-style software project**
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and this can be even used for something other than software build.

☐ **Build a maven2/3 project**
Build a maven 2/3 project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

☐ **Monitor an external job**
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. This is designed so that you can use Jenkins as a dashboard of your existing automation system. See [the documentation for more details](#).

☐ **Build multi-configuration project**
Suitable for projects that need a large number of different configurations, such as testing on multiple environments, platform-specific builds, etc.


☒ **Copy existing Job**
Copy from

Figure 13: Creation of a new build job in Jenkins (step 2).

The next screen offers an detailed configuration of the new job. Please check the name and the description of the project. The Section **Post-build Actions** must also be revised (cf. Figure 14 on the next page):

1. In **Publish Javadoc** and **Publish HTML reports** the existing project name must be replaced by the folder's name of the current project as defined in Bullet 1 on page 14.
2. Modify the **Record Emma coverage report** Section. Please insert the current coverage values to the cells. As a consequence, future builds with a worse coverage will be marked as unstable.
3. If other projects use this project, than edit also the **Build other projects** Section, remove this section otherwise.
4. Finally, edit the **Recipients** inside the **E-mail Notification** Section.

Post-build Actions


 **Publish Javadoc**

Javadoc directory

Directory relative to the root of the workspace, such as 'myproject/build/javadoc'

☐ Retain Javadoc for each successful build

Delete ?


 **Publish JUnit test result report**

Test report XMLs

[Fileset 'includes'](#) setting that specifies the generated raw XML report files, such as 'myproject/target/test-reports/*.xml'. Basedir of the fileset is [the workspace root](#).

☐ Retain long standard output/error



Delete ?



 **Record Emma coverage report**

Folders or files containing Emma XML reports


Specify the path to the Emmas XML report files, relative to [the workspace root](#).
- If you left this field blank the plugin will look for files matching the pattern: '**/emma/coverage*.xml' in the workspace.
- Or you can enclose the search specifying a list of files and folders separated by semicolon.
- Or use an Ant Fileset pattern.

Health reporting

	% Class	% Method	% Block	% Line	% Decision/Condition
	10	10	10	10	80
	8	8	8	8	0

Configure health reporting thresholds.
For the  row, leave blank to use the default values (i.e. 100, 70, 80, and 80 for class, method, block and line respectively).
For the  row, leave blank to use the default values (i.e. 0, 0, 0, 0).

Delete

 **Publish HTML reports**

HTML directory to archive


Index page[s]

Report title

Keep past HTML reports ☐

Add

Delete

 **E-mail Notification**

Recipients

Whitespace-separated list of recipient addresses. May reference build parameters like \$PARAM. E-mail will be sent when a build fails, becomes unstable or returns to stable.

☒ Send e-mail for every unstable build

☐ Send separate e-mails to individuals who broke the build

Save Apply Delete ?

Figure 14: Creation of a new build job in Jenkins (step 3).

4 Frequently Asked Questions

This section shows frequently recurring problems, their cause, and a solution to solve them. First, in Section 4.1 we present solutions for errors occurring during the compilation. Section 4.2 shows problems and solutions for errors, which may occur in testing.

4.1 Errors while compiling

This sections explains how build script errors can be resolved, which may appear while compiling projects and their plug-ins.

4.1.1 Compile failed

Description

The compilation target (`javac`) failed with an error message as shown below:

```
...
[javac] <path>\<class file>:88: error: cannot find symbol
[javac]         public void method(<a class> instance) {
[javac]                                ^
[javac]      symbol:   class <a class>
[javac]      location: class <class file>
[javac] <path>\<class file>:33: error: <some.package> does not exist
[javac] import <some.package>.<a class>;
[javac]         ^
[javac] <path>\<class file>:77: error: cannot find symbol
[javac]     return new <class name>();
[javac]                ^
[javac]      symbol:   class <class name>
[javac]      location: package <some.package>
[javac] 9 errors
[javac] 1 warning
```

BUILD FAILED

<project path>\build.xml:71: The following error occurred while executing this line:

<plug-in path>build-jk.xml:40: Compile failed; see the compiler error output for details.

Cause

The plug-in defining the missing classes and packages is not added correctly to the `includes` path element at the beginning of the file (c.f. Figure 7 on page 13). This can have multiple reasons:

1. The related plug-in is not added to the `includes` path element.
2. Some of the properties used inside the path element are pointing to the wrong location.

Solution

1. Add the necessary plug-ins to the `includes` path element as described in Section 3.1.1.
2. Check whether the properties are pointing to the correct locations, i.e. where the plug-in jars are created. You can add the following code at the beginning of the `compile` target to see the current content of the `includes` path element⁹:
`echo message="${toString:includes}"`

The printed result must not contain any ant variables in the form of

`${<a property name>}`.

You should also verify whether the properties of the `global-build.properties` are pointing to the correct location. The `libs` properties should point to the created jar file inside the corresponding project, i.e. `home` property. For instance:

`libs.model=${home.model.dir}/${build.jar.dir}/...varModel.jar`

⁹cf. comment by Alfonso Phocco at <http://www.jguru.com/faq/view.jsp?EID=471917>

4.2 Errors while testing

4.2.1 Could not find plug-in

Description

The automated test of the plug-in crashes with an error message as shown below:

```
org.eclipse.test.EclipseTestRunner$TestFailedException:
java.lang.Exception: Could not find plugin "de.uni-hildesheim.sse.<test plug-in name>"
    at org.eclipse.test.EclipseTestRunner.runFailed(EclipseTestRunner.java:435)
    at ...
```

Cause

The test plug-in could not be loaded by Eclipse. This means that either the test plug-in itself or one of the used plug-ins was not copied into the Eclipse test environment.

Solution

Find the missing plug-ins and add them to the `copy.to.eclipse` target (see Figure 10 on page 16, lines 32 - 35). The easiest way of doing so, is to run the build script for the complete project on a local machine, start the OSGi console of the Eclipse instance for testing, and try to load the test plug-in. This can be done as follows:

1. Open a console for the *plug-in's* folder of the Eclipse instance for testing. After running the build script, the Eclipse instance is located at the `testEclipse` folder inside the *project's* folder.
2. Open the OSGi console by running the command:
`java -jar org.eclipse.equinox.launcher_<version> -console`
An example is given in Figure 15.

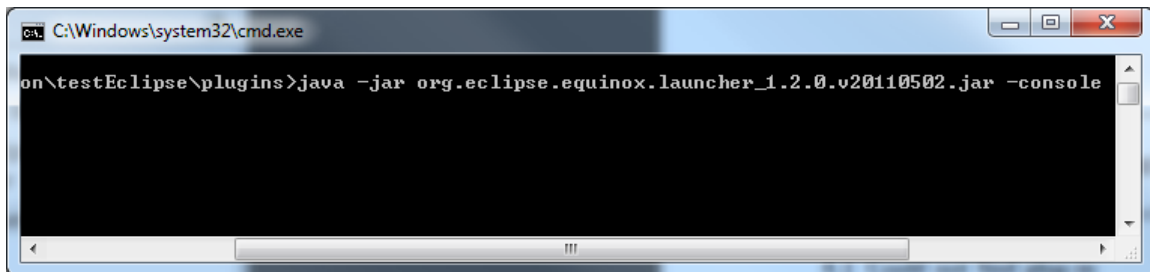
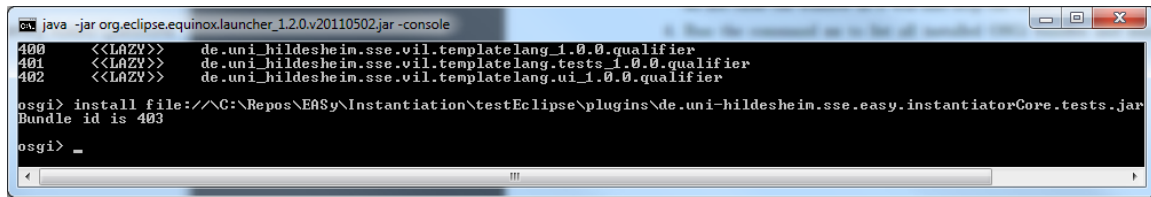


Figure 15: Command for opening the OSGi console. This must be done inside the *plug-in's* folder of the Eclipse instance for testing after running the build script.

3. Eclipse will ask for the workspace location. Ignore this window or accept it, but do not close the window as it will also stop the OSGi console.
4. Run the command `ss` to list all installed OSGi bundles and search the test plug-in.
5. If the plug-in is not listed, then install it with the command:
`install file://<location>`
Please note that you have to add a \ in front of the drive letter in Windows. An example is given in Figure 16 on the following page.



```

osgi> java -jar org.eclipse.equinox.launcher_1.2.0.v20110502.jar -console
400 <<LAZY>> de.uni_hildesheim.sse.vil.templatelang_1.0.0.qualifier
401 <<LAZY>> de.uni_hildesheim.sse.vil.templatelang.tests_1.0.0.qualifier
402 <<LAZY>> de.uni_hildesheim.sse.vil.templatelang.ui_1.0.0.qualifier
osgi> install file:///C:/Repos/EASy/Instantiation/testEclipse/plugins/de.uni-hildesheim.sse.easy.instantiatorCore.tests.jar
Bundle id is 403
osgi> _

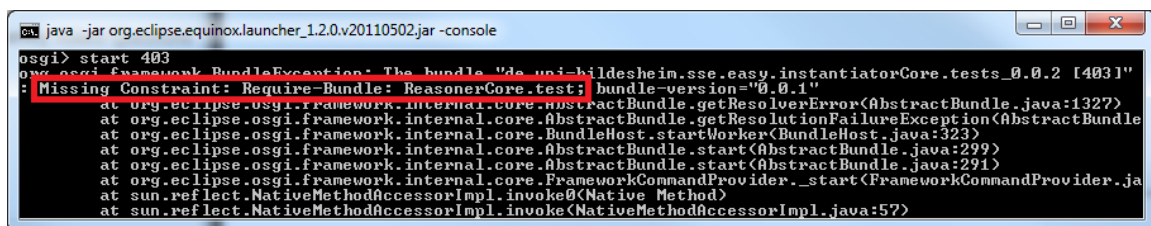
```

Figure 16: Installation of a plug-in via the OSGi console. In Windows, you have to add a \ in front of the drive letter.

6. Try to load the bundle with running the command:

`start <plug-in number>`

This should fail, but the displayed error message will give you a hint, which plug-in is missing. Figure 17 shows the useful error message.



```

osgi> java -jar org.eclipse.equinox.launcher_1.2.0.v20110502.jar -console
osgi> start 403
org.osgi.framework.BundleException: The bundle "de.uni-hildesheim.sse.easy.instantiatorCore.tests_0.0.2 [403]"
: Missing Constraint: Require-Bundle: ReasonerCore.test; bundle-version="0.0.1"
at org.eclipse.osgi.framework.internal.core.AbstractBundle.getResolverError(AbstractBundle.java:1327)
at org.eclipse.osgi.framework.internal.core.AbstractBundle.getResolutionFailureException(AbstractBundle
at org.eclipse.osgi.framework.internal.core.BundleHost.startWorker(BundleHost.java:323)
at org.eclipse.osgi.framework.internal.core.AbstractBundle.start(AbstractBundle.java:299)
at org.eclipse.osgi.framework.internal.core.AbstractBundle.start(AbstractBundle.java:291)
at org.eclipse.osgi.framework.internal.core.FrameworkCommandProvider._start(FrameworkCommandProvider.ja
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)

```

Figure 17: Failed attempt of starting a plug-in via the OSGi console. The error message shows that the *ReasonerCore.tests* plug-in could not be started.

7. Check whether the crashing plug-in is located inside the *plug-in*'s folder of the Eclipse instance for testing. Repeat the steps above, if the plug-in is located inside the *plug-in*'s folder. Add the missing plug-in to the `copy.to.eclipse` target (see Figure 10 on page 16, lines 32 - 35).

4.2.2 java.lang.NoSuchMethodError: com.vladium.emma.rt.RT.S

Description

The automated test of the plug-in crashes with an error message as shown below:

```
java.lang.NoSuchMethodError: com.vladium.emma.rt.RT.S([[Ljava/lang/String;J)V
    at de.uni_hildesheim.sse.parserantlr.internal.InternalVilBuildLanguage...
    at ...
```

Cause

The specified class was not instrumented correctly. This can have multiple reasons:

1. The Java file contains longer `static final String[]` definitions, which are not supported by Emma (cf. <http://sourceforge.net/p/emma/bugs/96/>).

Solution

Depending on the cause, different solutions exist:

1. In case a Java file contains longer `static final String[]` definitions, exclude this file from instrumentation. For this purpose, you have to add an exclusion filter to the `instrument` target (cf. Figure 18).

```
1 <target name="instrument">
2   <emma enabled="true" verbosity="verbose">
3     <instr destdir="${coverage.instr.dir}" metadatafile="${coverage.
4       results.dir}/metadata.emma" mode="fullcopy" >
5       <filter excludes="de.uni_hildesheim.sse.parserantlr.internal.*"
6         />
7       <instrpath>
8       </instrpath>
9     </instr>
10  </emma>
11 </target>
```

Figure 18: Modified Build script (build.xml) of the Instantiation project (excerpt). A new exclusion filter was specified in line 4 to exclude all Java classes of the `de.uni_hildesheim.sse.parserantlr.internal` package to avoid `java.lang.NoSuchMethodError: com.vladium.emma.rt.RT.S` error.

4.2.3 java.lang.NoClassDefFoundError: Could not initialize class <class name>

Description

The automated test of the plug-in crashes with an error message as shown below:

```
java.lang.NoClassDefFoundError: Could not initialize class <class name>
    at java.lang.reflect.Constructor.newInstance(Constructor.java:525)
    at ...
```

Cause

This problem has several possible causes:

1. This problem is related with the problem described in Section 4.2.2 on the preceding page.
2. This problem also appears if referenced libraries are not packed into the plug-in's jar file. In this case, <class name> should be the name of a third party class, e.g. org.apache.commons.io/FilenameUtils.

Solution

1. Solve the problem as described in Section 4.2.2.
2. Edit the build-jk file of the plug-in and add the copy task of Figure 19 to the jar target.

```
1 <!-- Creates a jar file -->
2 <target name="jar" depends="compile">
3   <copy todir="${build.classes.dir}" failonerror="true" overwrite="
4     true">
5     <fileset dir="${basedir}">
6       <include name="lib/**/*" />
7     </fileset>
8   </copy>
9   <jar destfile="${build.jar.dir}/${ant.project.name}.jar" basedir="
    ${build.classes.dir}" manifest="META-INF/MANIFEST.MF" />
  </target>
```

Figure 19: Modified Build script (build-jk.xml) of de.uni_hildesheim.sse.vil.build-lang.tests plug-in (excerpt). The lines 3 – 7 are added to include necessary libraries.